Programming Guidelines for NPARC Alliance Software Development*

Charles E. Towne
NASA Glenn Research Center
Cleveland, Ohio

1 Introduction

This document describes the programming guidelines to be used during NPARC Alliance software development projects[1]. It deals almost exclusively with Fortran 77[2], since most NPARC Alliance software is written in that language, but some parts are also applicable to C.

The guidelines are intended to enhance the following aspects of the final product, listed in decreasing order of importance:

- **Maintainability** — refers to how easy it is to understand the purpose of each element of the program, and to modify and extend the program.
- **Portability** — refers to how easily the program can be ported to new computational platforms.
- **Efficiency** — refers to the amount of computer resources (CPU time, memory, disk storage, etc.) required to run the program.

Much of this material has been taken, sometimes verbatim, from the following documents available on the World-Wide Web:

- Levine, David L., "Fortran 77 Coding Guidelines," http://www.aeem.iastate.edu/Computers/Software/Fortran/guide.html
- Huddleston, John, "Fortran Coding Standard"
- Robey, Tom, and Smith, Brian, eds., "Future of Fortran - Moving from F77"

2 Program Development and Design

Items in this section are fairly general and fundamental in nature. They impact all three of the items listed above — maintainability, portability, and efficiency.

*This is version 1.0 of this document, released 26 Feb 1997.

[1]Since the capability requirements list for the NASTD/NPARC/NXAIR code merger effort includes a "grandfather clause" allowing (in fact, encouraging) the reuse of existing code, the programming guidelines described in this document will not necessarily be followed throughout that final product. They should, however, be followed for any new code that is written. And, where possible without adversely impacting the development schedule, existing code should be "cleaned up" to be consistent these guidelines.

[2]Throughout this document, the term "Fortran" should be understood to mean Fortran 77.

2.1 Language

Use ANSI standard Fortran 77 exclusively, with the following exceptions:

- ANSI C code may be used where Fortran 77 is inadequate, such as for dynamic memory allocation and interfacing with PVM. Try to minimize the amount of C code, however.
- In addition to the standard Fortran character set, the characters `!` and `&` may be used, as well as the lower-case alphabetic characters.
- The following non-standard, but widely available, extensions to Fortran 77 are allowed:
  - `Include`
  - `Implicit none`
  - `Do ... Enddo`
  - In-line comments
  - Variable, parameter, subprogram, and common block names as long as eight characters
- Before the new code is formally released, it will be tested using *ftnchek* to confirm adherence to the Fortran 77 standard (except as noted above), and, where possible, to flag code that doesn't follow the programming guidelines described in this document.

2.2 Organization

- Write modular code.
- In general, put each subprogram in a separate file, using the subprogram name as the file name, with a `.f` extension.
- Group related files in a single directory, but don't go overboard.
- Names of files and directories should reflect their purpose.

2.3 Common Blocks

- Don't use blank common.
- Put all common blocks in include files, one per file, using the common block name as the file name, with a `.inc` extension.
- Strike a common-sense balance on the number of common blocks. Group related variables together in separate common blocks, but minimize the total number of blocks.
- `Save` all common blocks, in the include file.

2.4 Data Types

- Use `Implicit none` in each program unit, and explicitly declare all variables and parameters. Common variables and parameters should be declared in the relevant include file.
- Use the following data types:

  - `Character[*`n`]`
  - `Complex`
  - `Double Precision`
  - `Integer`
  - `Logical`
  - `Real`

- Don't use `*`'ed forms, like `Real*8`.
- Don't compare arithmetic expressions of different types; convert the type explicitly.

2.5 Compilation

- Use a makefile for routine compiling and linking.
- Avoid embedding compiler directives in the code, unless an equivalent compiler option is not available. In that case, include comments describing its effect, the target machine and operating system, and the compiler.
- During development, low-level-optimization, no-optimization, and/or special debugging compiler options are often used. Before releasing a code modification, re-test it with the same level of optimization that will be used for the final production code.
- Before releasing a code modification, re-test it using compiler options that flag various run-time error conditions. Examples include options to detect attempts to use variables that haven't been assigned a value, or arrays with out-of-range indices.

3 Coding Style

Items in this section are fairly specific, and primarily impact the readability, and thus the maintainability, of the final product. It is recognized that rules for "good coding style" are somewhat subjective.[3] Some flexibility for personal preference should be acceptable. However, these guidelines should be followed at least in spirit throughout the program.

3.1 Program Units

- Begin main programs with a `Program` statement.
- Don't use multiple entries or alternate returns.
- Order subprogram arguments as: input variables, control variables, output variables, external names.
- Match the arguments in the calling (sub)program to those of the called subprogram in both number and type.
- Use the following order for statements within each subprogram:

[3] The guidelines in this document no doubt reflect some of my own biases.

    - Standard header section
    - Parameter definitions
    - Common blocks
    - Type statements for subprogram arguments
    - Type statements for local variables
    - Executable code

- Functions should not have side effects. (I.e., don't change the arguments or any common variables inside the function.)
- Use generic names for library functions, rather than precision-specific ones.
- Name external functions in an `External` statement.

3.2 Statement Form

- Use standard fixed-form formatting — labels in columns 1–5, continuation character in column 6, statement itself in columns 7–72.
- Use a `c` in column 1 for non-blank comment lines.
- Use a `&` in column 6 for continuation lines.
- Split long lines before or after an operator, preferably a `+` or `-`.
- Don't write more than one statement per line.

3.3 Statement Labels

- Don't use unreferenced labels.
- Use labels in ascending order.
- Use label values, in conjunction with comments, that visually divide code into logical sections.

3.4 Upper/Lower Case

- Use upper case for parameters, lower case with an initial capital letter for Fortran keywords, and lower case for everything else except comments.
- Write comments as normal text, with normal capitalization rules.

3.5 Spacing

- Use spacing to enhance readability.
- Indent contents of do loops and if blocks — suggested amount is three spaces.
- Don't use tabs.

- Use spacing in equations to clarify precedence of operators. I.e., normally put one space on either side of `=`, `+`, and `-` operators (except in subscripts), but none around `*`, `/`, or `**` operators. For example, this:

```
y1 = (-b + Sqrt(b**2 - 4.*a*c))/(2.*a)
```

  is easier to read than this:

```
y1=(-b+Sqrt(b**2-4.*a*c))/(2.*a)
```

  or this:

```
y1 = ( - b + Sqrt ( b ** 2 - 4. * a * c ) ) / ( 2. * a )
```

- Use spacing to reveal patterns in continuation lines and in separate but logically related statements. For example, this:

```
 dum1 = Sqrt((fr (i,j) - fr ( 1, j))**2 +
&            (fth(i,j) - fth( 1, j))**2)
 dum2 = Sqrt((fr (i,j) - fr (n1, j))**2 +
&            (fth(i,j) - fth(n1, j))**2)
 dum3 = Sqrt((fr (i,j) - fr ( i, 1))**2 +
&            (fth(i,j) - fth( i, 1))**2)
 dum4 = Sqrt((fr (i,j) - fr ( i,n2))**2 +
&            (fth(i,j) - fth( i,n2))**2)
```

  is easier to read than this:

```
 dum1 = Sqrt((fr(i,j) - fr(1,j))**2 + (fth(i,j) -
&fth(1,j))**2)
 dum2 = Sqrt((fr(i,j) - fr(n1,j))**2 + (fth(i,j) -
&fth(n1,j))**2)
 dum3 = Sqrt((fr(i,j) - fr(i,1))**2 + (fth(i,j) -
&fth(i,1))**2)
 dum4 = Sqrt((fr(i,j) - fr(i,n2))**2 + (fth(i,j) -
&fth(i,n2))**2)
```

3.6 Variable Names

- Use letters and digits only, with the first character a letter.
- In general, follow standard Fortran convention for the variable type. I.e., integers start with `i`, `j`, `k`, `l`, `m`, or `n`, all others are real.
- Use names that are descriptive of the entity being represented, and/or are consistent with the standard notation in the field.
- When several different entities are stored in an array, like the `biga` array in NPARC, define and use Fortran parameters with descriptive names for the subscript values that correspond to the various entities in the array.
- Don't use keyword, subprogram, or common block names for variables.
- Don't give a local variable the same name as a common variable.

3.7 Arrays

- Dimension arrays in the type statement, not in the common block or a separate `Dimension` statement.
- When passing one-dimensional arrays and character variables into a subprogram, use the assumed-length form for the array declarator and character type statement inside the subprogram. I.e.,

```
Subroutine sub (x,c)
Dimension x(*)
Character*(*) c
```

- Don't exceed the bounds of the array dimensions.
- Specify all the subscripts when referencing an array.
- Begin array indices with 1, unless there's a *very* good (and well-commented) reason to do otherwise.

3.8 Common Blocks

- To assure proper word boundary alignment, order variables in common blocks as double precision/logical/real/integer.
- Don't include common blocks in subprograms where they aren't needed.

3.9 Fortran Parameters

- Use `Parameter` statements to symbolically name constants that may change from compilation to compilation, or that are long and susceptible to typing errors.
- Use `Parameter` statements to symbolically name array dimensions that may change from compilation to compilation.
- `Parameter` statements should be in a separate include file.

3.10 Control Statements

- Minimize the use of `Goto` statements, especially where they can be replaced by short'ish if blocks, but don't create convoluted code just to avoid using them. Don't be afraid to use a `Goto` where it makes sense. An example might be a long (more than a page) conditional section of code. In this case a well-commented `Goto` block, which ends with an easily-noticed statement label, may be more readable than an indented if block without an ending statement label. Also consider making a long conditional section a separate subprogram.
- For long do loops (more than a page), use conventional do loops, ending with a labeled `Continue` statement.
- End conventional do loops with distinct `Continue` statements. I.e., don't end nested loops with the same `Continue` statement.
- Don't jump into the middle of a do loop (i.e., no extended-range do loops) or an if block.

3.11 Comments

- Use comments liberally to describe what's being done. Where code may be confusing, use longer comments to describe why something's being done the way that it is.
- Make each comment meaningful; don't simply re-iterate what's already obvious from the coding itself. As an obvious example, this:

```
c-----Get the turbulent viscosity using Baldwin-Lomax model
      Call turbbw
```

  is more meaningful than this:

```
c-----Call turbbw
      Call turbbw
```

- Use a consistent method to help the reader distinguish comments from code, such as the "`---`" leaders in the examples above.
- Start the text of comments at the same indentation level as the code being described.
- Use a standard header section at the beginning of each subprogram defining its purpose.
- Use in-line comments, with `!` as the delimiter, where appropriate for short explanations or clarifications. Start in-line comments far enough to the right (e.g., three spaces or more from the end of the statement) to help distinguish comments from code. Where appropriate, align them vertically with nearby in-line comments.
- Define each common block variable using an in-line comment on its type statement in the include file. Each common variable will thus have a separate type statement.
- Define key local variables using in-line comments on the type statements in the subprogram.

3.12 Input/Output

- Open output files with `Status='unknown'` where possible.
- Use input error recovery parameters such as `End=`, `Err=`, and `Iostat=`.
- Place once-used `Format` statements immediately following their reference. Place those used more than once at the end of the subprogram.

3.13 Obsolete/Forbidden Features

The following Fortran features are either formally declared as obsolete, or widely considered to be poor programming practice:

- Arithmetic if statements.
- Do loops with real indices.
- `Pause` statements.

- `Assign` and assigned `Goto` statements.
- Hollerith edit descriptors and Hollerith character strings.
- `Equivalence` statements. If used, document their purpose with a comment.

Appendix — Standard Subprogram Format

The following is an example illustrating the use of the guidelines in this document for one of the subprograms in NASTD. First, here's an include file named `test.inc` that's needed:

```
      Common /test/ itest
      Integer itest(200)   ! Test options
      Save test
```

And here's the subprogram itself, named `blomax.f`:

```
      Subroutine blomax (jedge,re,yy,rh,uu,vo,vi,tauw,tv)
c
c-----Purpose:  This subroutine computes the turbulent viscosity
c               coefficient using the Baldwin-Lomax model.
c
      Implicit none
c-----Parameter statements
c-----Common blocks
      Include 'test.inc'
c-----Input arguments
      Integer jedge   ! Index of boundary layer edge
      Real re         ! Reference Reynolds number
      Real yy(*)      ! Distance from wall
      Real rh(*)      ! Static density
      Real uu(*)      ! Velocity
      Real vo(*)      ! Vorticity
      Real vi(*)      ! Laminar viscosity coefficient
      Real tauw       ! Shear stress at the wall
c-----Output arguments
      Real tv(*)      ! Turbulent viscosity coefficient
c-----Local variables
      Integer icross  ! Flag for inner/outer region
      Integer j       ! Do loop index
      Integer jedg    ! Index limit for Fmax search
      Real al         ! Mixing length
      Real aplus      ! Van Driest damping constant
      Real bigk       ! Clauser constant
      Real ccp        ! Constant in outer region model
      Real ckleb      ! Constant in Klebanoff intermittency factor
      Real cwk        ! Constant in outer region model
      Real fkleb      ! Klebanoff intermittency factor
      Real fl         ! Baldwin-Lomax F parameter
      Real fmax       ! Baldwin-Lomax Fmax parameter
```

```
      Real frac       ! Fractional decrease in F req'd for finding peak
      Real fwake      ! Baldwin-Lomax Fwake parameter
      Real rdum       ! Ratio of distance from wall to ymax
      Real smlk       ! Von Karman constant
      Real tvi        ! Inner region turbulent viscosity coefficient
      Real tvo        ! Outer region turbulent viscosity coefficient
      Real udif       ! Max velocity difference
      Real umax       ! Max velocity
      Real umin       ! Min velocity
      Real ymax       ! Distance from wall to location of Fmax
      Real yp         ! y+
      Real ypcon      ! Coefficient term for y+, based on wall values
      Real ypconl     ! Coefficient term for y+
      Real yyj        ! Distance from wall
c
c-----Set constants
c
      aplus = 26.
      ccp   = 1.6
      ckleb = 0.3
      cwk   = 0.25
      smlk  = 0.4
      bigk  = 0.0168
      If (itest(126) .eq. 1) bigk = 0.0180   ! Comp. correction (cfl3de)
c
c-----Compute stuff needed in model
c
c-----Get coefficient term for y+
      If (itest(25) .eq. 1) Then   ! Using wall vorticity as in cfl3de
         ypcon = Sqrt(re*rh(1)*vo(1)/vi(1))
      Else                         ! Using wall shear stress
         If (tauw .le. 1.e-9) tauw = 1.e-9
         ypcon = Sqrt(re*rh(1)*tauw)/vi(1)
      End if
c-----Set index limit for Fmax search, and fractional decrease needed to
c-----   qualify as first peak
      jedg = jedge
      frac = .70
      If (itest(163) .gt. 0) Then        ! User-spec. frac. decrease
         frac = Real(itest(163))/1000.
      Else if (itest(163) .lt. 0) Then   ! Reset search range, use max
         jedg = Min(jedge,-itest(163))   !    value, not first peak
         frac = 0.0
      Endif
c-----Get max velocity and max velocity difference
      umin = 0.
      umax = 0.
      Do 100 j = 2,jedge
         umax = Max(umax,uu(j))
         umin = Min(umin,uu(j))
  100 Continue
```

```
      udif = umax - umin
c
c-----Get Fmax by searching for first peak in F
c
      ymax = 0.
      fmax = 0.
      ypconl = ypcon
      Do 200 j = 2,jedg
         yyj = yy(j)
         If (itest(26) .eq. 1) Then   ! Use local values in y+
            ypconl = ypcon*Sqrt(rh(j)/rh(1))*vi(1)/vi(j)
         End if
         yp = ypconl*yyj   ! y+
         fl = yyj*vo(j)*(1. - Exp(-yp/aplus))   ! B-L F parameter
         If (fl .gt. fmax) Then             ! Set new Fmax
            fmax = fl
            ymax = yyj
         Else if (fl .gt. frac*fmax) Then   ! Keep searching
         Else                               ! Found Fmax, so get out
            Go to 210
         End if
  200 Continue
c-----Reset ymax and Fmax if necessary, to avoid overflows
  210 If (ymax .lt. 1.e-6) ymax = 5.e-5
      If (fmax .lt. 1.e-6) fmax = 5.e-5
c
c-----Compute turbulent viscosity
c
      icross = 0
      ypconl = ypcon
      Do 300 j = 2,jedge
         yyj = yy(j)
         tvi = 1.e10
c--------Inner region value, if we're still there
         If (icross .eq. 0) Then
            If (itest(26) .eq. 1) Then   ! Use local values in y+
               ypconl = ypcon*Sqrt(rh(j)/rh(1))*vi(1)/vi(j)
            End if
            yp = ypconl*yyj   ! y+
            al = smlk*yyj*(1. - Exp(-yp/aplus))   ! Mixing length
            tvi = rh(j)*al*al*vo(j)
         End if
c--------Outer region value
         rdum = yyj/ymax
         If (rdum .ge. 1.e5) Then   ! Prevent overflow
           fkleb = 0.0
         Else                       ! Klebanoff intermittency factor
           fkleb = 1./(1. + 5.5*(ckleb*rdum)**6)
         End if
         fwake = Min(ymax*fmax,cwk*ymax*udif*udif/fmax)
         tvo = bigk*ccp*rh(j)*fwake*fkleb
```

```
c--------Set turbulent viscosity, plus flag if we're in outer region
         tv(j) = tvi
         If (tvo .lt. tvi) Then
            icross = 1
            tv(j) = tvo
         End if
c--------Non-dimensionalize
         tv(j) = re*tv(j)
  300 Continue
c-----Zero out turbulent viscosity at wall
      tv(1) = 0.0
      Return
      End
```